本节内容

# 二叉排序树

# (BST)

# 知识总览

- 二叉排序树
  - 二叉排序树的定义
  - 查找操作
  - 插入操作
  - 删除操作
  - 查找效率分析

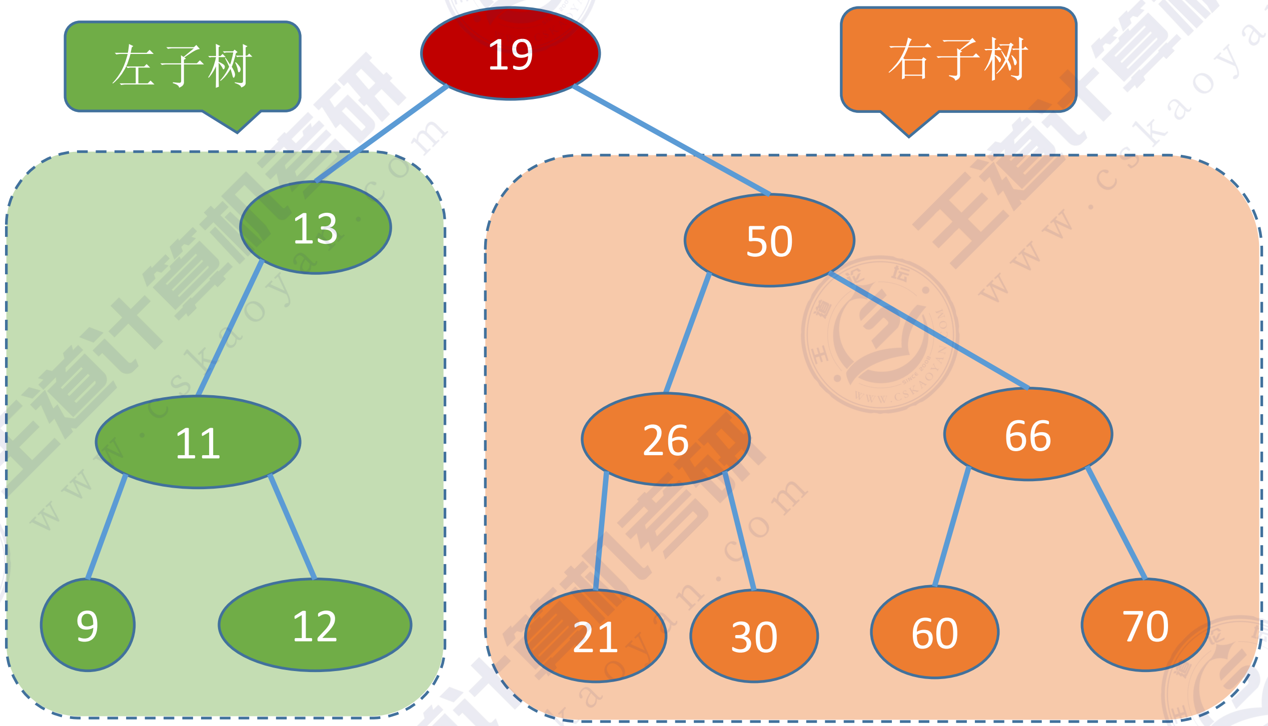
# 二叉排序树的定义

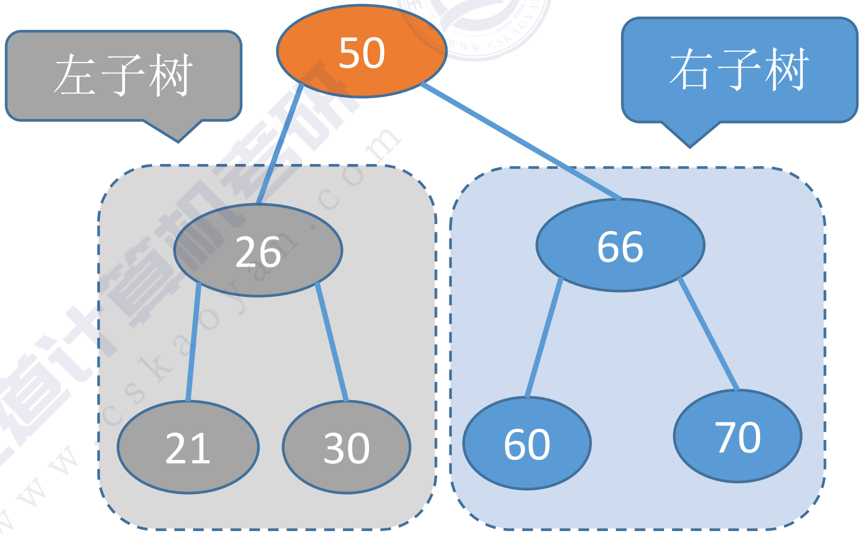
二叉排序树可用于元素的有序组织、搜索

二叉排序树，又称二叉查找树（BST，Binary Search Tree）
一棵二叉树或者是空二叉树，或者是具有如下性质的二叉树：
左子树上所有结点的关键字均小于根结点的关键字；
右子树上所有结点的关键字均大于根结点的关键字。
左子树和右子树又各是一棵二叉排序树。

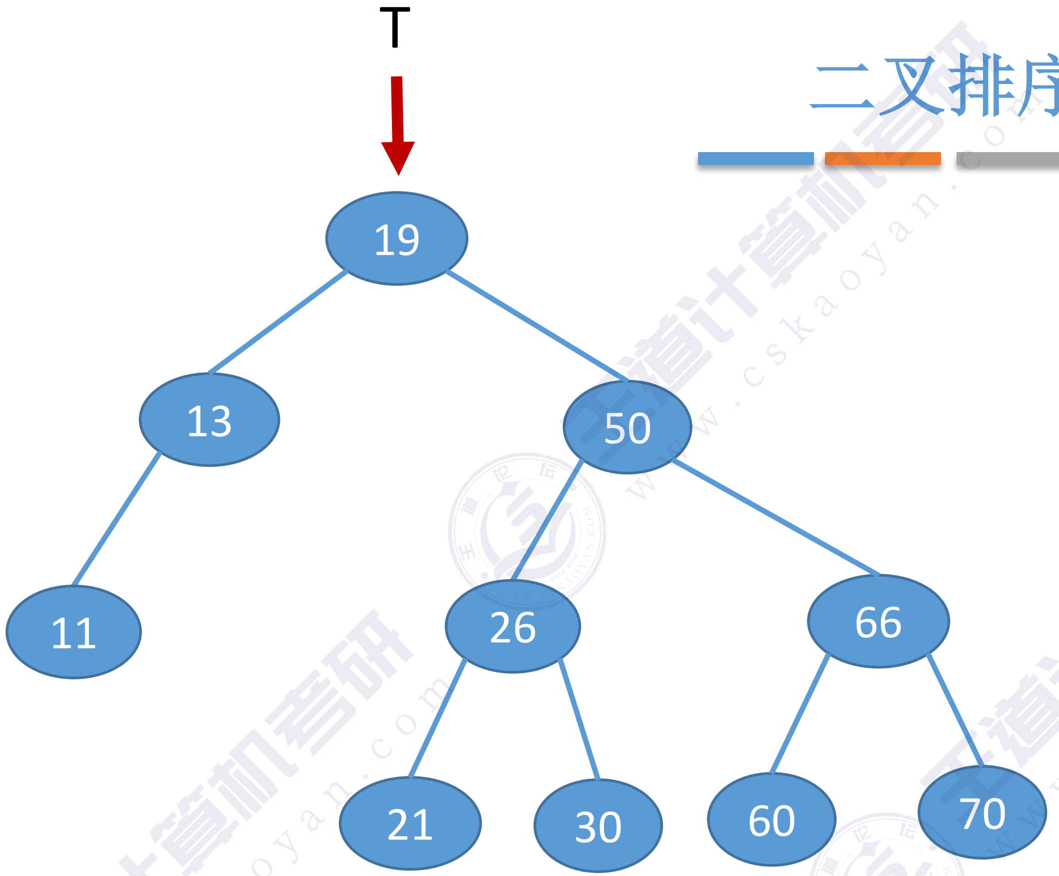
左子树结点值 < 根结点值 < 右子树结点值

进行中序遍历，可以得到一个递增的有序序列



# 二叉排序树的查找




左子树结点值 < 根结点值 < 右子树结点值

若树非空，目标值与根结点的值比较：
若相等，则查找成功；
若小于根结点，则在左子树上查找，否则在右子树上查找。

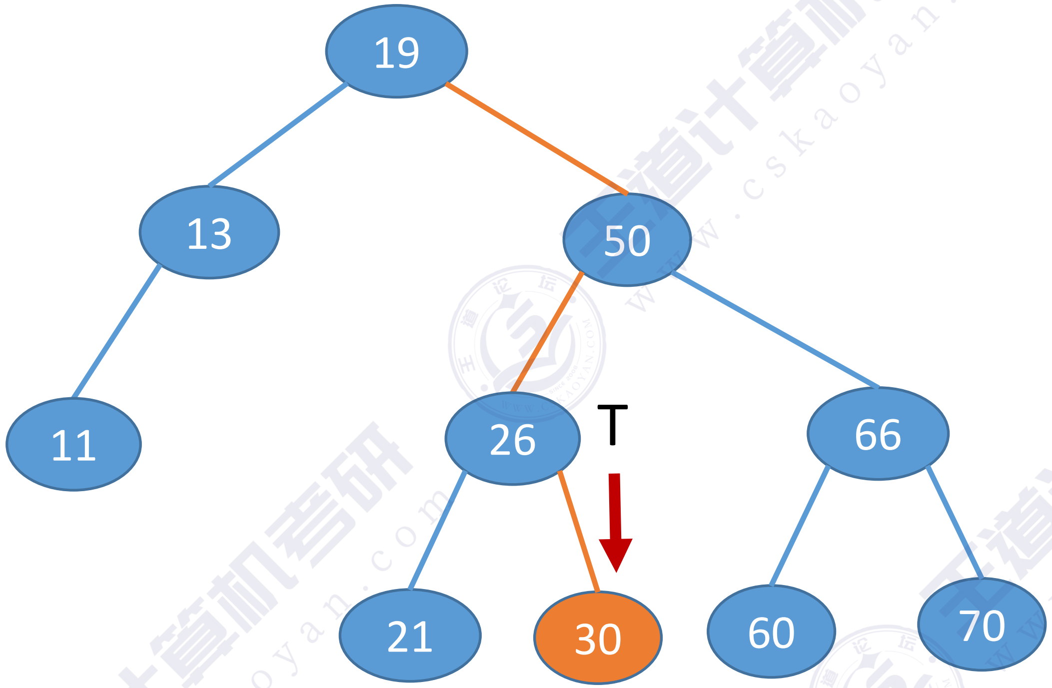
查找成功，返回结点指针；查找失败返回NULL

例1：查找关键字为30的结点

```
//二叉排序树结点
typedef struct BSTNode{
    int key;
    struct BSTNode  *lchild,*rchild;
}BSTNode,*BSTree;
```

```
//在二叉排序树中查找值为 key 的结点
BSTNode *BST_Search(BSTree T,int key){
    while(T!=NULL&&key!=T->key){     //若树空或等于根结点值，则结束循环
        if(key<T->key) T=T->lchild;     //小于，则在左子树上查找
        else  T=T->rchild;              //大于，则在右子树上查找
    }
    return T;
}
```

# 二叉排序树的查找




左子树结点值 < 根结点值 < 右子树结点值

若树非空，目标值与根结点的值比较：
若相等，则查找成功；
若小于根结点，则在左子树上查找，否则在右子树上查找。

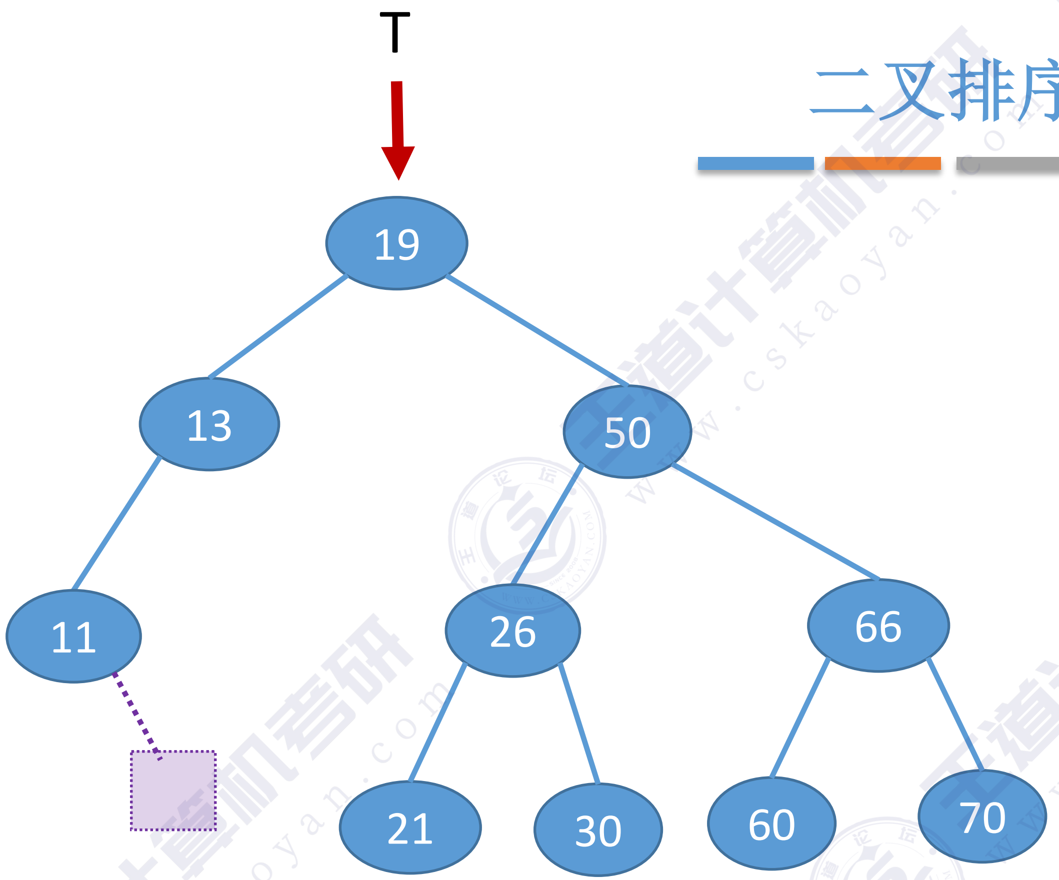
查找成功，返回结点指针；查找失败返回NULL

例1：查找关键字为30的结点

```
//二叉排序树结点
typedef struct BSTNode{
    int key;
    struct BSTNode  *lchild,*rchild;
}BSTNode,*BSTree;
```

```
//在二叉排序树中查找值为 key 的结点
BSTNode *BST_Search(BSTree T,int key){
    while(T!=NULL&&key!=T->key){     //若树空或等于根结点值，则结束循环
        if(key<T->key) T=T->lchild;     //小于，则在左子树上查找
        else   T=T->rchild;              //大于，则在右子树上查找
    }
    return T;
}
```

# 二叉排序树的查找




左子树结点值 < 根结点值 < 右子树结点值

若树非空，目标值与根结点的值比较：
若相等，则查找成功；
若小于根结点，则在左子树上查找，否则在右子树上查找。

查找成功，返回结点指针；查找失败返回NULL

例2：查找关键字为12的结点

```
//二叉排序树结点
typedef struct BSTNode{
    int key;
    struct BSTNode  *lchild,*rchild;
}BSTNode,*BSTree;
```

```
//在二叉排序树中查找值为 key 的结点
BSTNode *BST_Search(BSTree T,int key){
    while(T!=NULL&&key!=T->key){      //若树空或等于根结点值，则结束循环
        if(key<T->key) T=T->lchild;       //小于，则在左子树上查找
        else   T=T->rchild;               //大于，则在右子树上查找
    }
    return T;
}
```

# 二叉排序树的查找

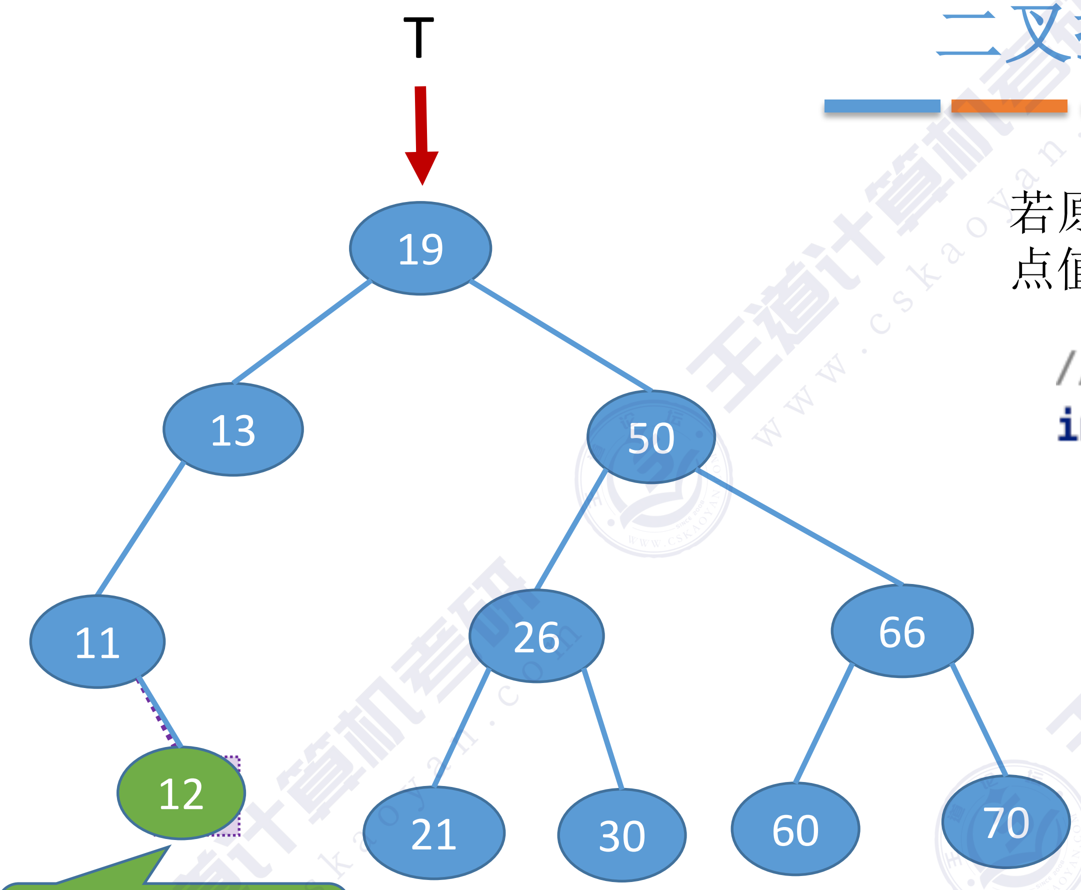
```
//在二叉排序树中查找值为 key 的结点
BSTNode *BST_Search(BSTree T,int key){
    while(T!=NULL&&key!=T->key){     //若树空或等于根结点值，则结束循环
        if(key<T->key) T=T->lchild;      //小于，则在左子树上查找
        else  T=T->rchild;              //大于，则在右子树上查找
    }
    return T;
}
```

最坏空间复杂度O(1)

```
//在二叉排序树中查找值为 key 的结点（递归实现）
BSTNode *BSTSearch(BSTree T,int key){
    if (T==NULL)
        return NULL;     //查找失败
    if (key==T->key)
        return T;        //查找成功
    else if (key < T->key)
        return BSTSearch(T->lchild, key);   //在左子树中找
    else
        return BSTSearch(T->rchild, key);   //在右子树中找
}
```

最坏空间复杂度O(h)

# 二叉排序树的插入




新插入的结点一定是叶子

例：插入关键字为12的结点

若原二叉排序树为空，则直接插入结点；否则，若关键字k小于根结点值，则插入到左子树，若关键字k大于根结点值，则插入到右子树

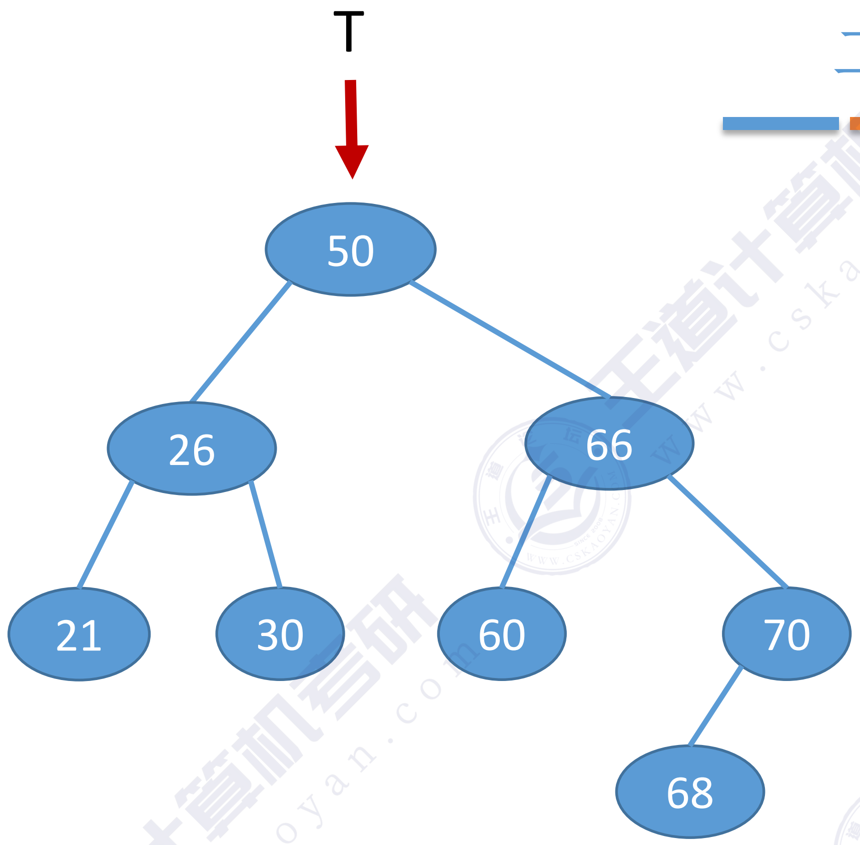
最坏空间复杂度O(h)

```
//在二叉排序树插入关键字为k的新结点（递归实现）
int BST_Insert(BSTree &T, int k){
    if(T==NULL){                    //原树为空，新插入的结点为根结点
        T=(BSTree)malloc(sizeof(BSTNode));
        T->key=k;
        T->lchild=T->rchild=NULL;
        return  1;                  //返回1，插入成功
    }
    else if(k==T->key)              //树中存在相同关键字的结点，插入失败
        return  0;
    else if(k<T->key)               //插入到T的左子树
        return  BST_Insert(T->lchild,k);
    else                            //插入到T的右子树
        return  BST_Insert(T->rchild,k);
}
```

嗨嗨，醒醒，敲代码了！

练习：实现非递归插入

# 二叉排序树的构造



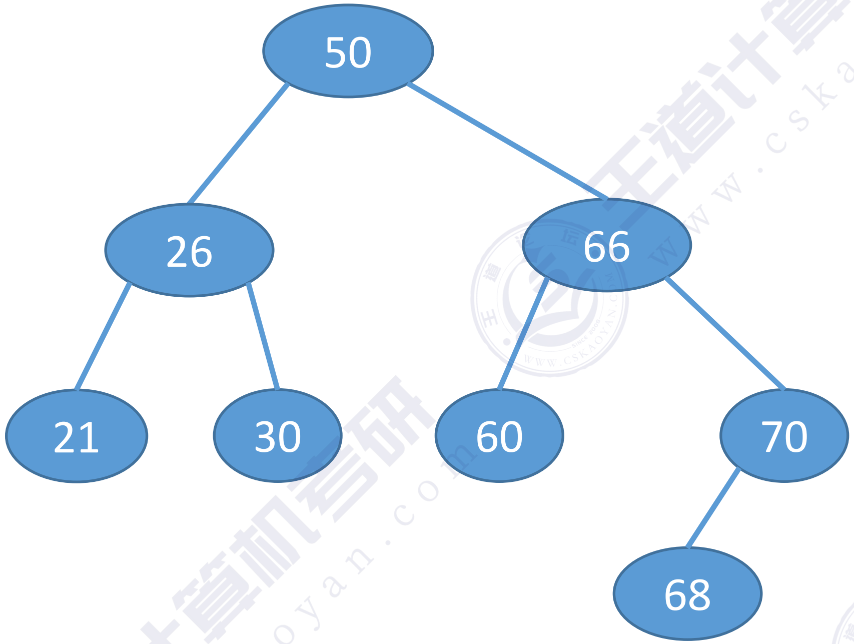

```
//按照 str[] 中的关键字序列建立二叉排序树
void Creat_BST(BSTree &T,int str[],int n){
    T=NULL;             //初始时T为空树
    int i=0;
    while(i<n){         //依次将每个关键字插入到二叉排序树中
        BST_Insert(T,str[i]);
        i++;
    }
}
```

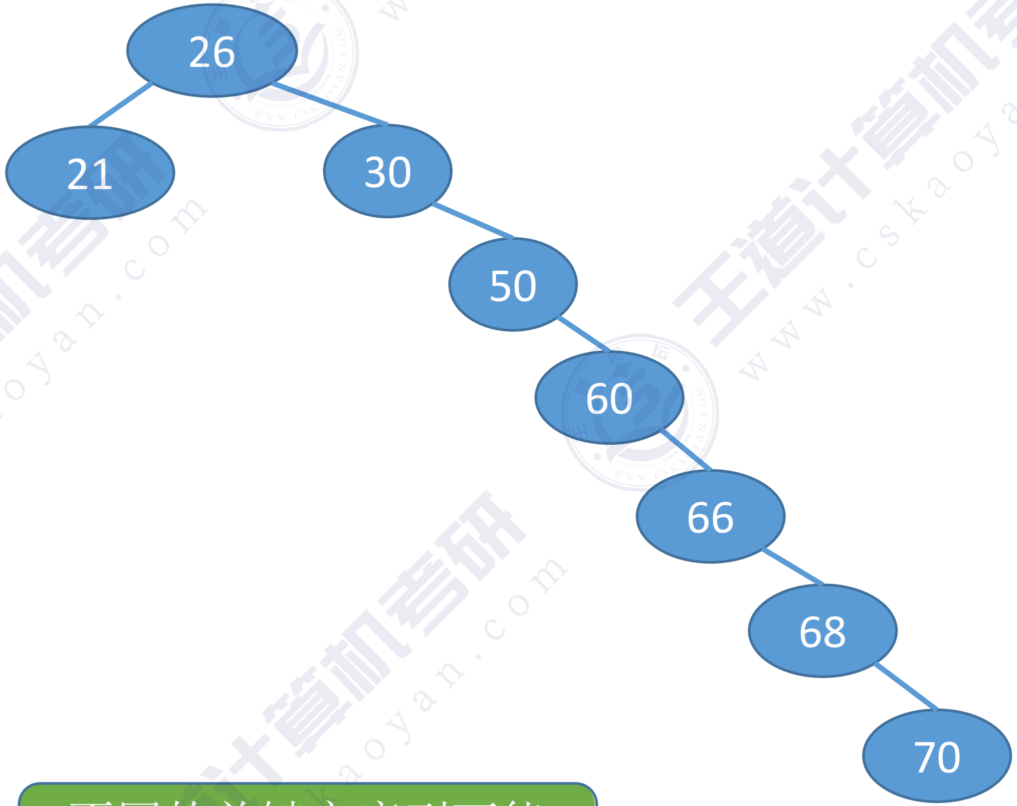
例1：按照序列str={50, 66, 60, 26, 21, 30, 70, 68}建立BST

例2：按照序列str={50, 26, 21, 30, 66, 60, 70, 68}建立BST

不同的关键字序列可能得到同款二叉排序树

# 二叉排序树的构造




例1：按照序列str={50, 66, 60, 26, 21, 30, 70, 68}建立BST

例2：按照序列str={50, 26, 21, 30, 66, 60, 70, 68}建立BST

例3：按照序列str={26, 21, 30, 50, 60, 66, 68, 70}建立BST

不同的关键字序列可能得到同款二叉排序树

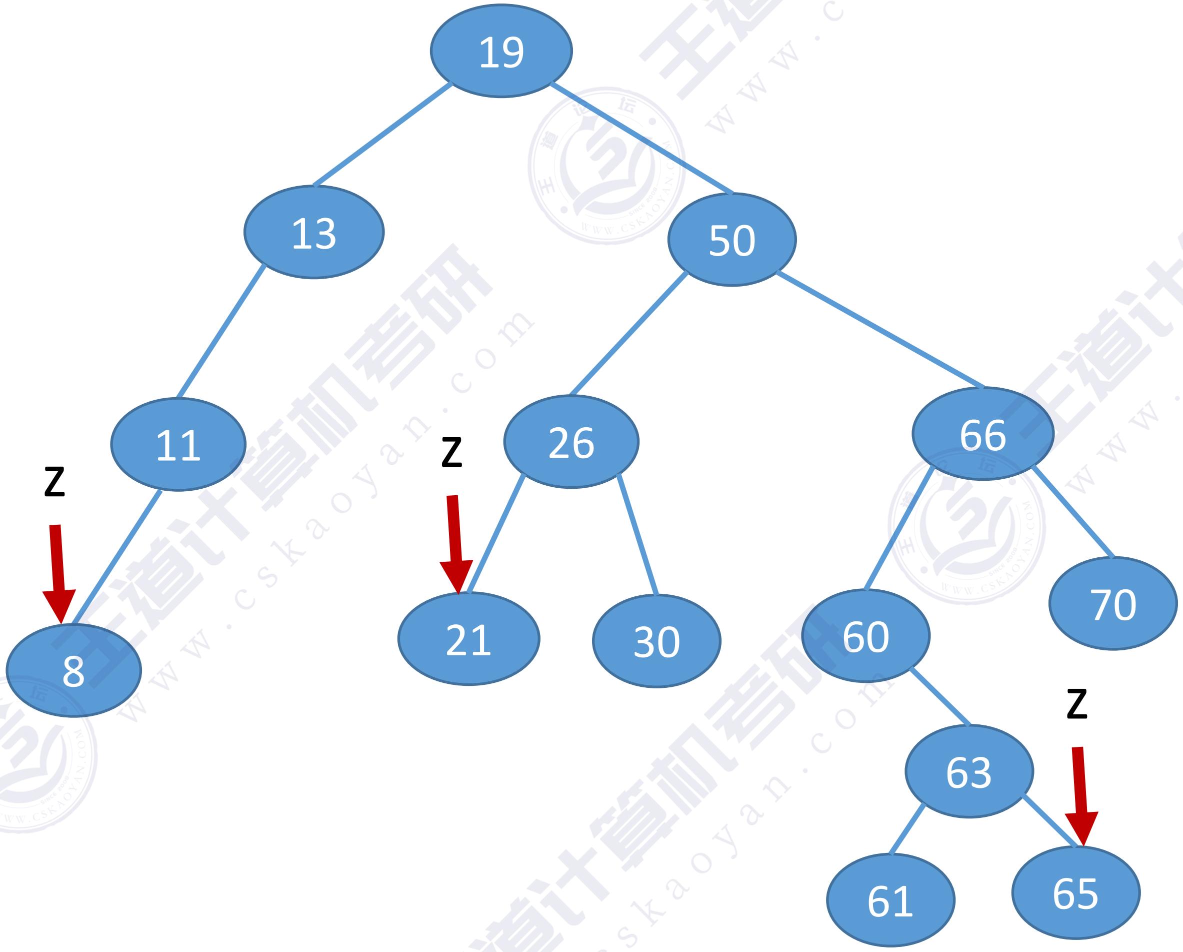
也可能得到不同款二叉排序树

# 二叉排序树的删除

先搜索找到目标结点：

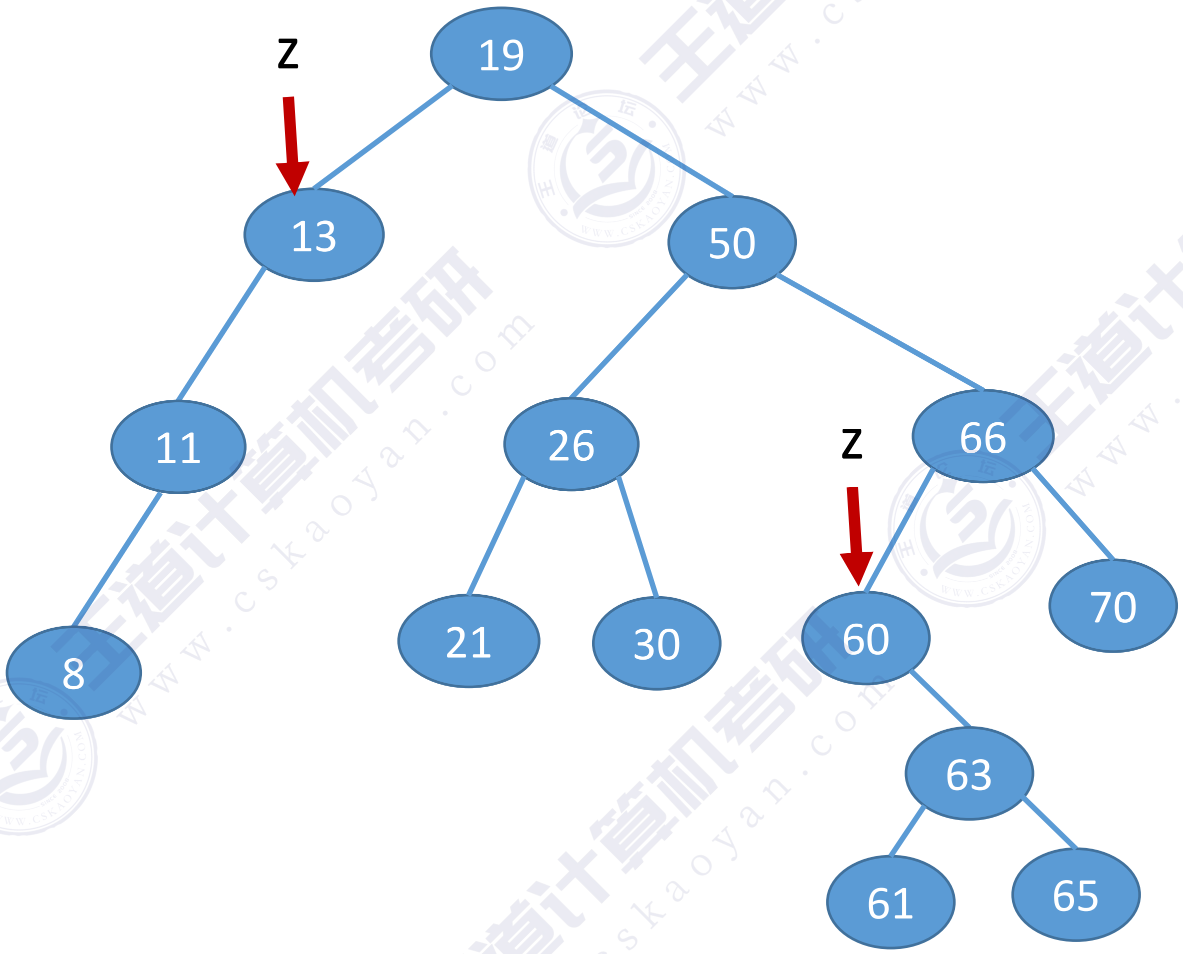
① 若被删除结点*z*是叶结点，则直接删除，不会破坏二叉排序树的性质。

左子树结点值 < 根结点值 < 右子树结点值



# 二叉排序树的删除

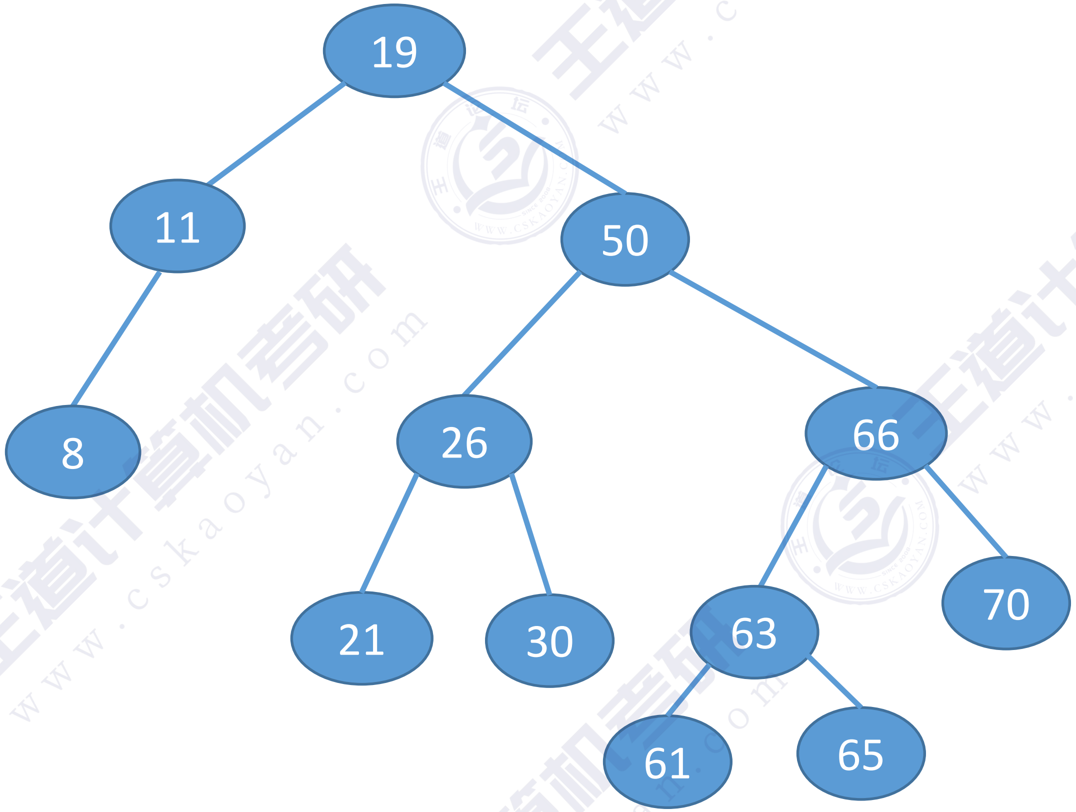
② 若结点z只有一棵左子树或右子树，则让z的子树成为z父结点的子树，替代z的位置。




左子树结点值 < 根结点值 < 右子树结点值

# 二叉排序树的删除

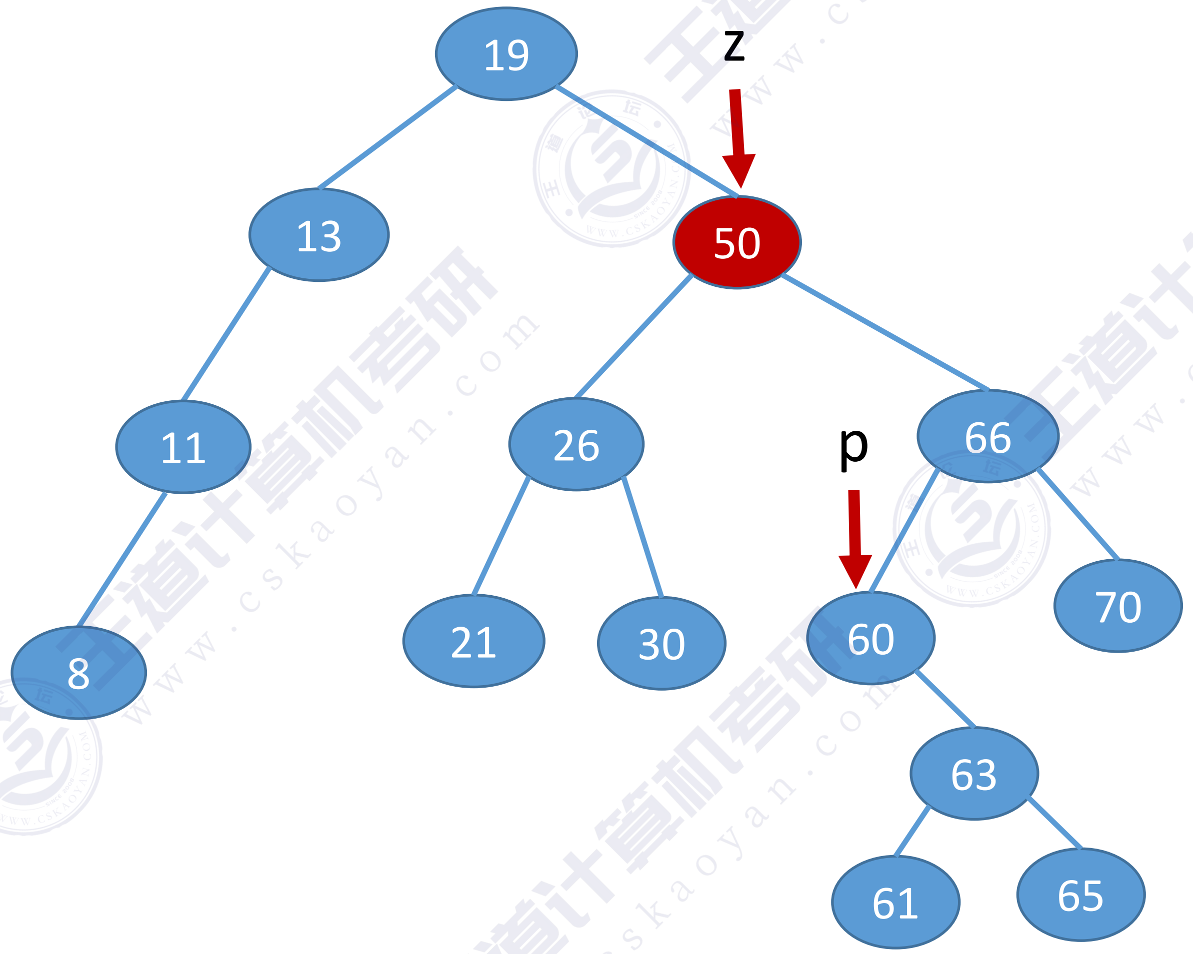
② 若结点z只有一棵左子树或右子树，则让z的子树成为z父结点的子树，替代z的位置。




左子树结点值 < 根结点值 < 右子树结点值

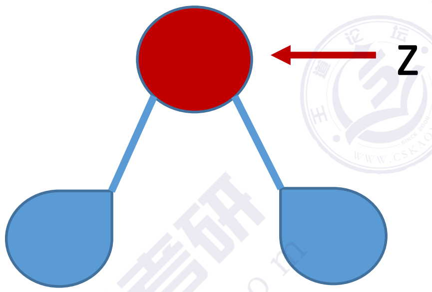
# 二叉排序树的删除

③ 若结点*z*有左、右两棵子树，则令*z*的直接后继（或直接前驱）替代*z*，然后从二叉排序树中删去这个直接后继（或直接前驱），这样就转换成了第一或第二种情况。




左子树结点值 < 根结点值 < 右子树结点值

进行中序遍历，可以得到一个递增的有序序列




中序遍历——左　根　右

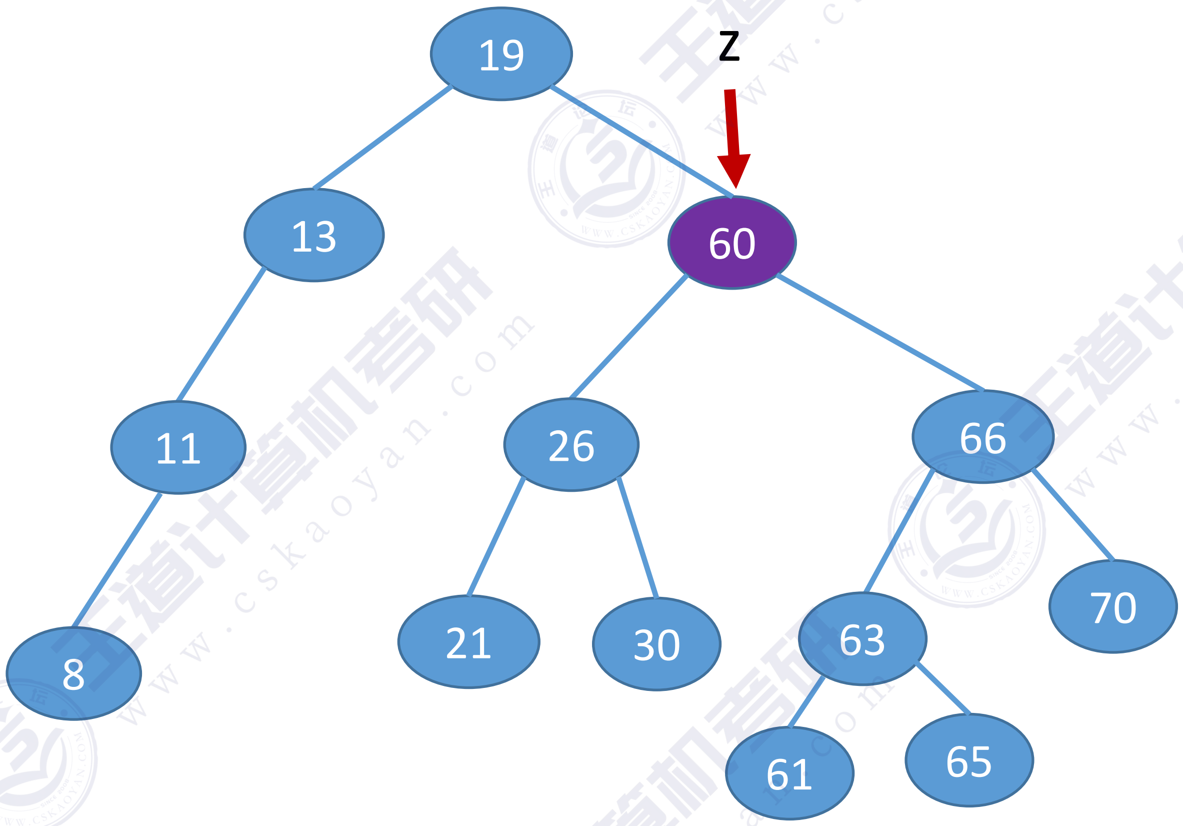
左　根　(左　根　右)

左　根　((左　根　右)　根　右)

z的后继：z的右子树中最左下结点（该节点一定没有左子树）

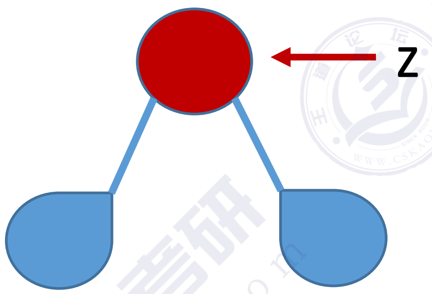
# 二叉排序树的删除

③ 若结点z有左、右两棵子树，则令z的直接后继（或直接前驱）替代z，然后从二叉排序树中删去这个直接后继（或直接前驱），这样就转换成了第一或第二种情况。




左子树结点值 < 根结点值 < 右子树结点值

进行中序遍历，可以得到一个递增的有序序列




中序遍历——左　根　右

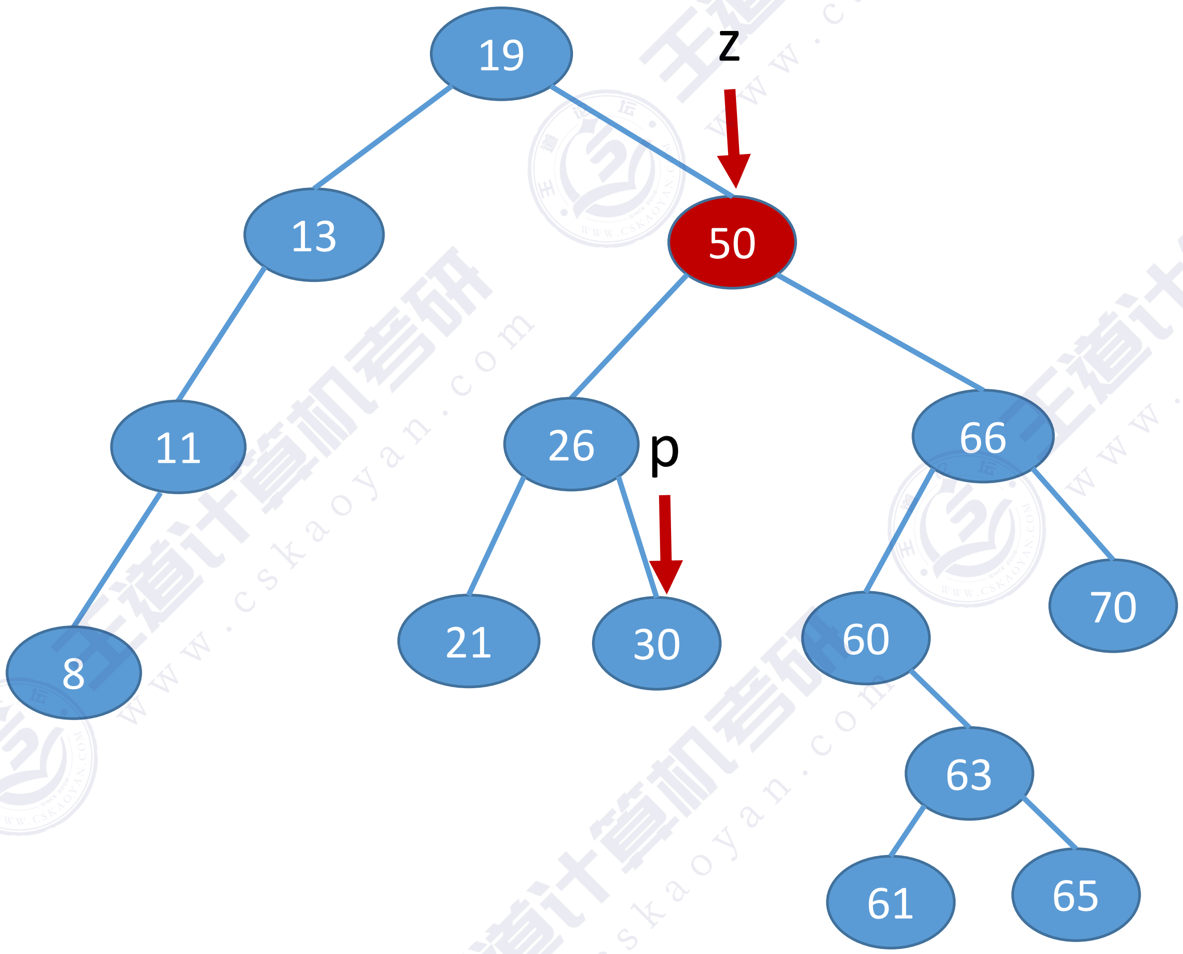
左　根　(左　根　右)

左　根　((左　根　右)　根　右)

z的后继：z的右子树中最左下结点（该节点一定没有左子树）

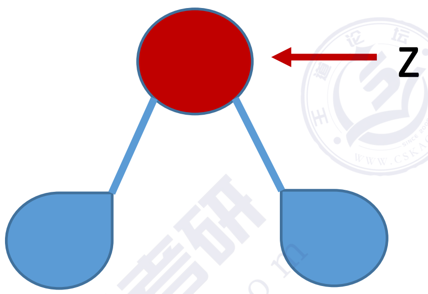
# 二叉排序树的删除

③ 若结点z有左、右两棵子树，则令z的直接后继（或直接前驱）替代z，然后从二叉排序树中删去这个直接后继（或直接前驱），这样就转换成了第一或第二种情况。




左子树结点值 < 根结点值 < 右子树结点值

进行中序遍历，可以得到一个递增的有序序列




中序遍历——左 根 右

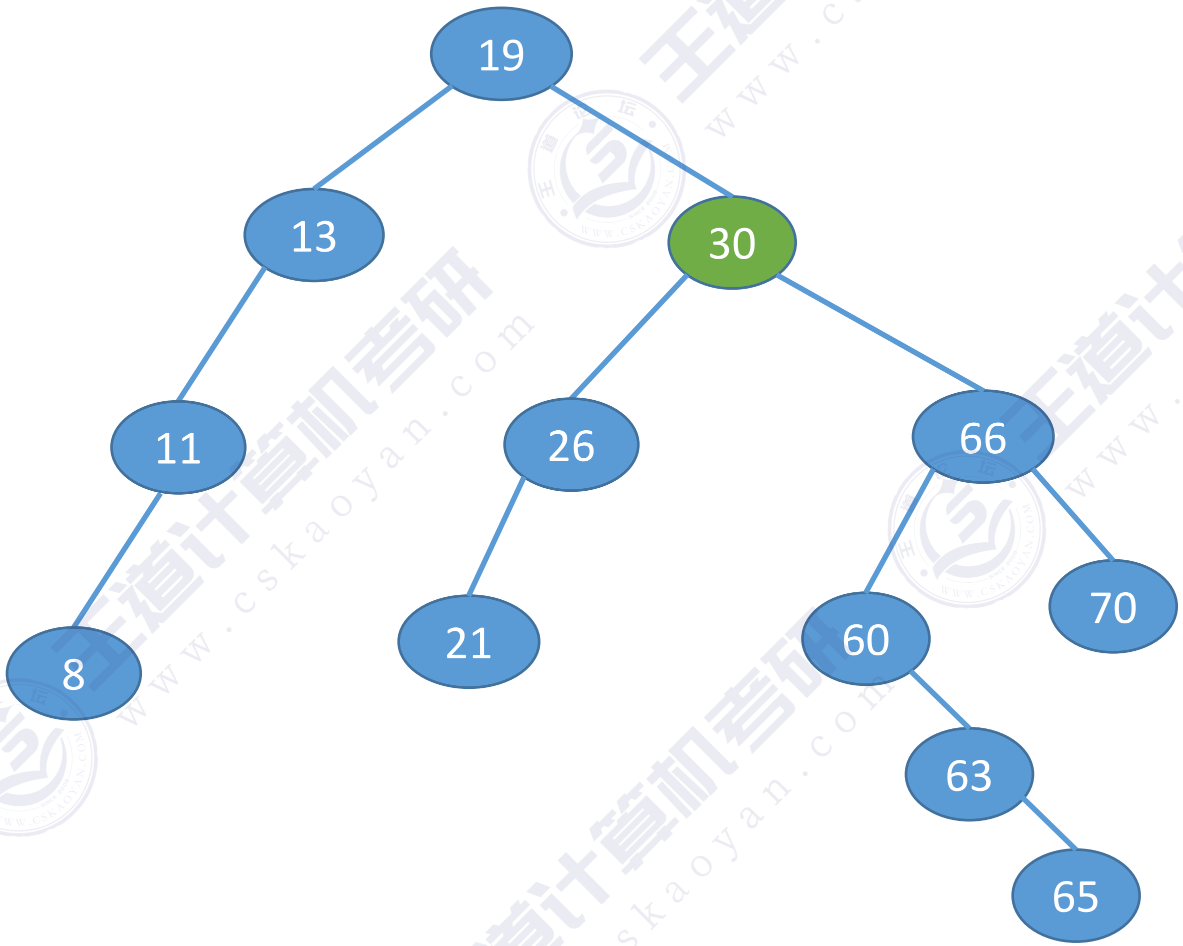
(左 根 右) 根 右

(左 根 (左 根 右)) 根 右

z的前驱：z的左子树中最右下结点（该节点一定没有右子树）

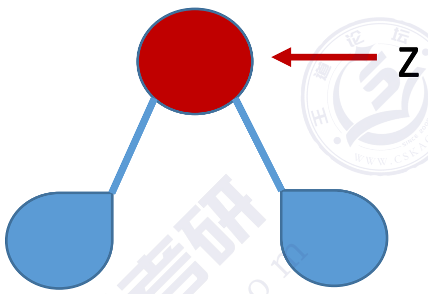
# 二叉排序树的删除

③ 若结点z有左、右两棵子树，则令z的直接后继（或直接前驱）替代z，然后从二叉排序树中删去这个直接后继（或直接前驱），这样就转换成了第一或第二种情况。




左子树结点值 < 根结点值 < 右子树结点值

进行中序遍历，可以得到一个递增的有序序列




中序遍历——左　根　右

(左　根　右)　根　右

(左　根　(左　根　右))　根　右

z的前驱：z的左子树中最右下结点（该节点一定没有右子树）

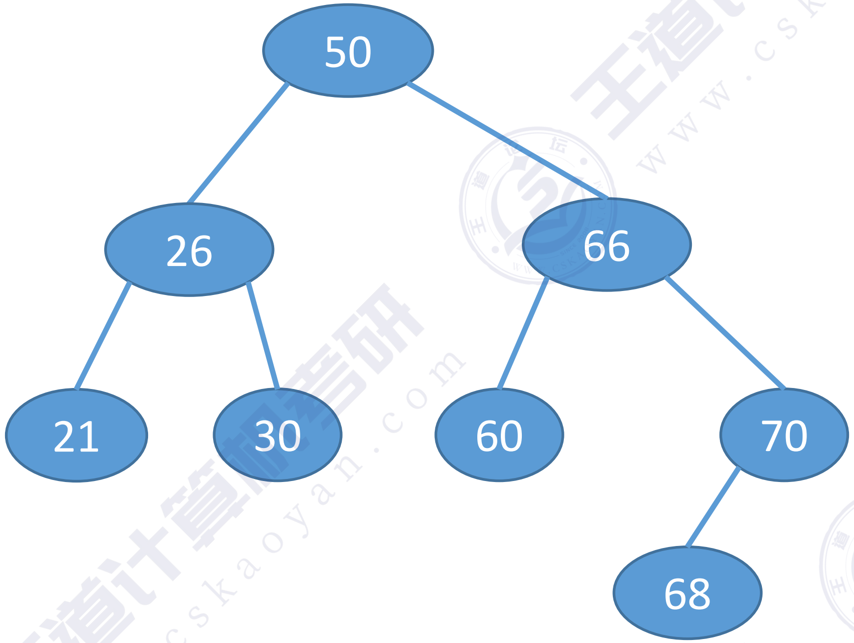
若树高h，找到最下层的一个结点需要对比 h 次

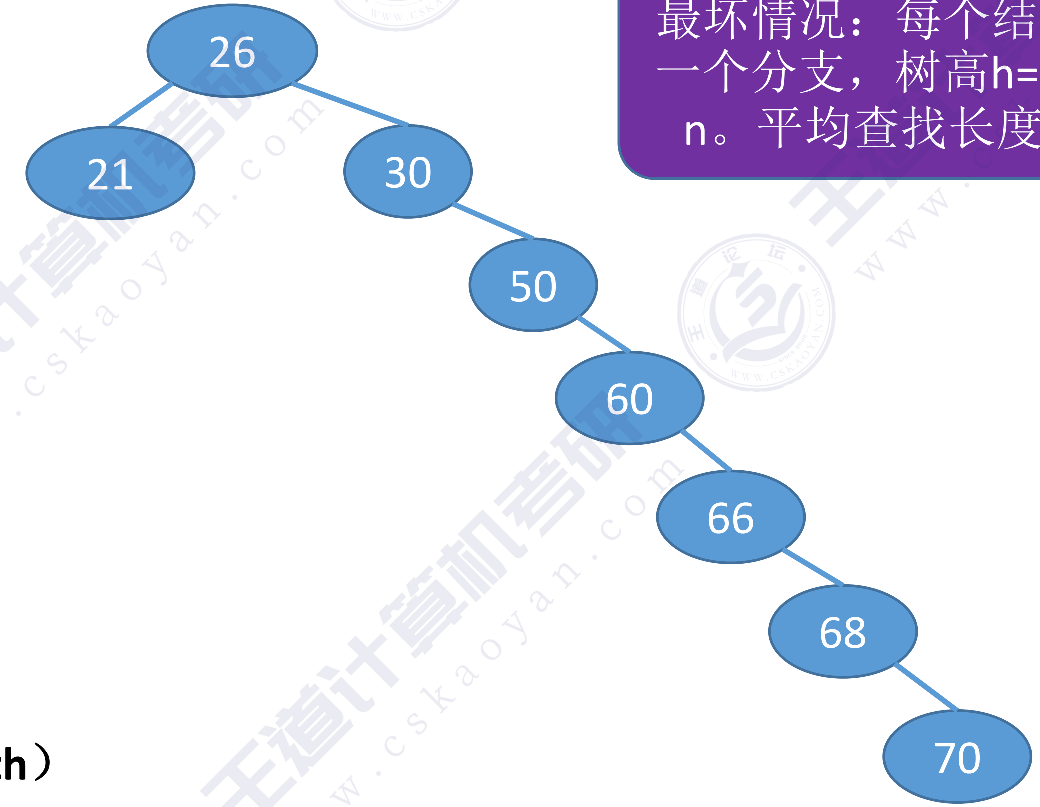
# 查找效率分析

最好情况：n个结点的二叉树最小高度为$\lfloor \log_2 n \rfloor + 1$。平均查找长度= $O(\log_2 n)$

查找长度——在查找运算中，需要对比关键字的次数称为查找长度，反映了查找操作时间复杂度

最坏情况：每个结点只有一个分支，树高h=结点数n。平均查找长度=$O(n)$



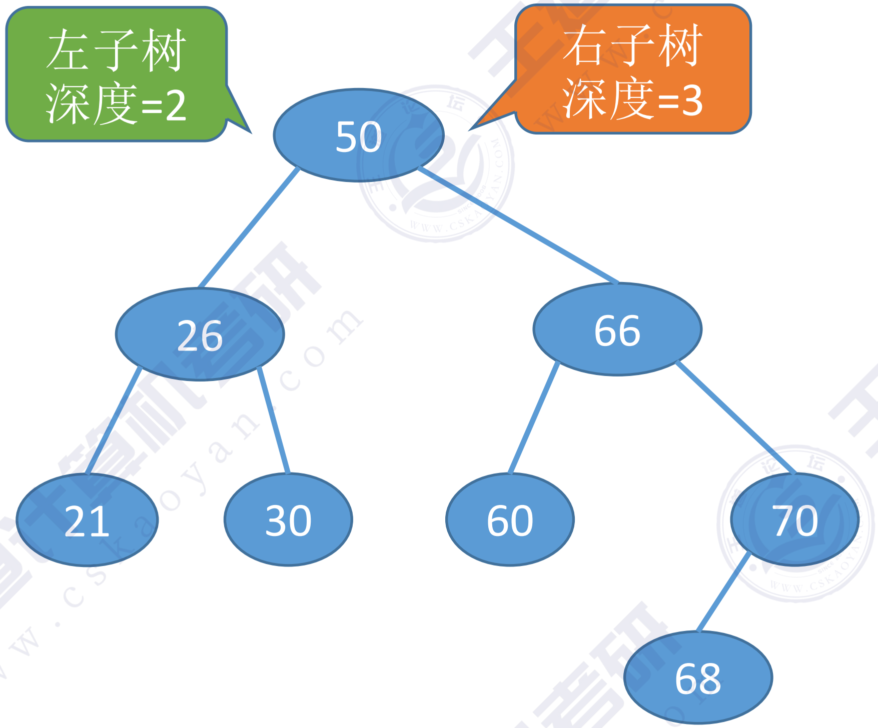

**查找成功**的**平均**查找长度 **ASL**（**Average Search Length**）

ASL = (1*1 + 2*2 + 3*4 + 4*1)/8 = 2.625

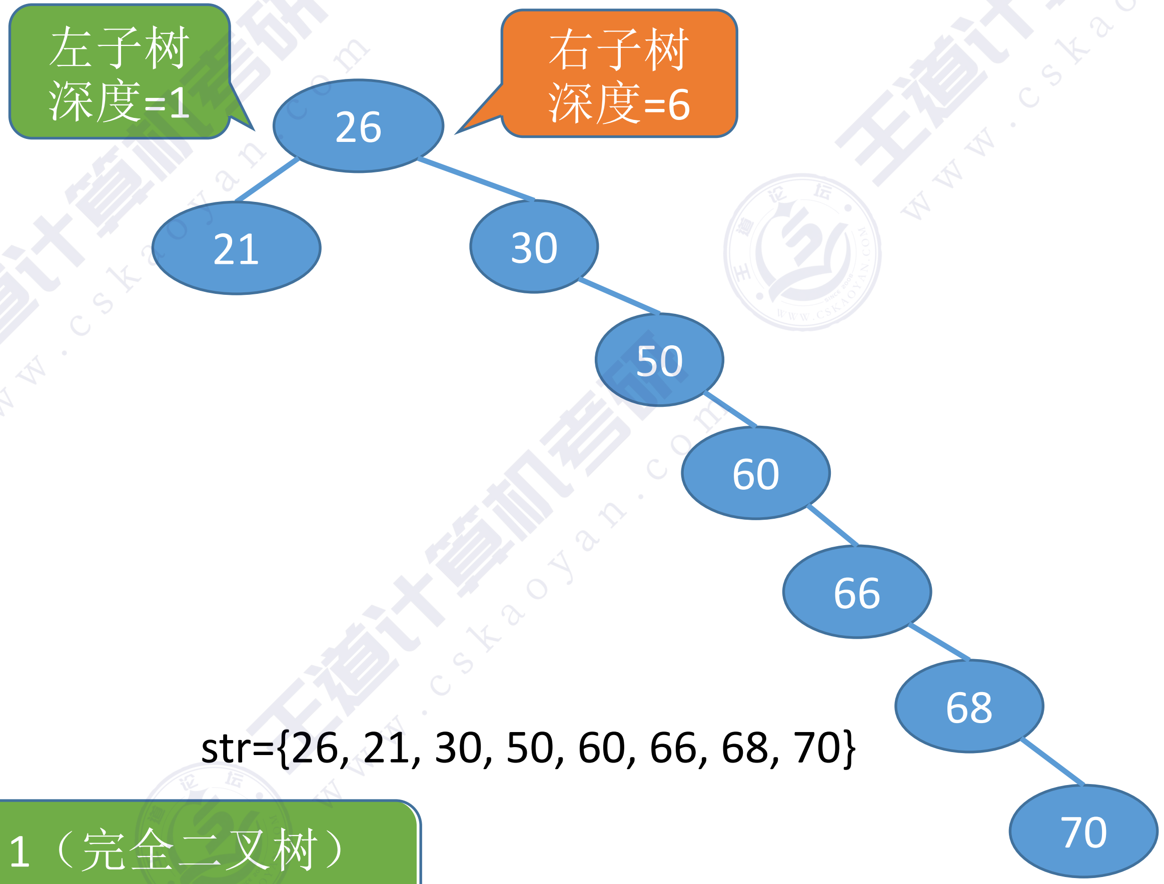
ASL = (1*1 + 2*2 + 3*1 + 4*1 + 5*1 + 6*1+ 7*1)/8 = 3.75

# 查找效率分析

平衡二叉树。树上任一结点的左子树和右子树的深度之差不超过1。




str={50, 66, 60, 26, 21, 30, 70, 68}

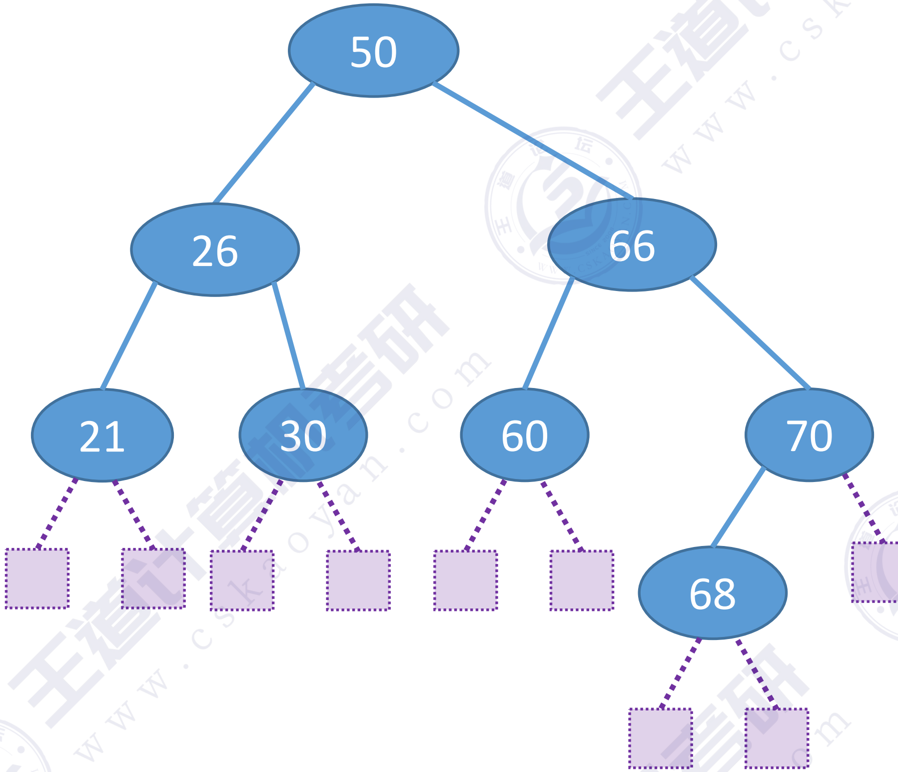


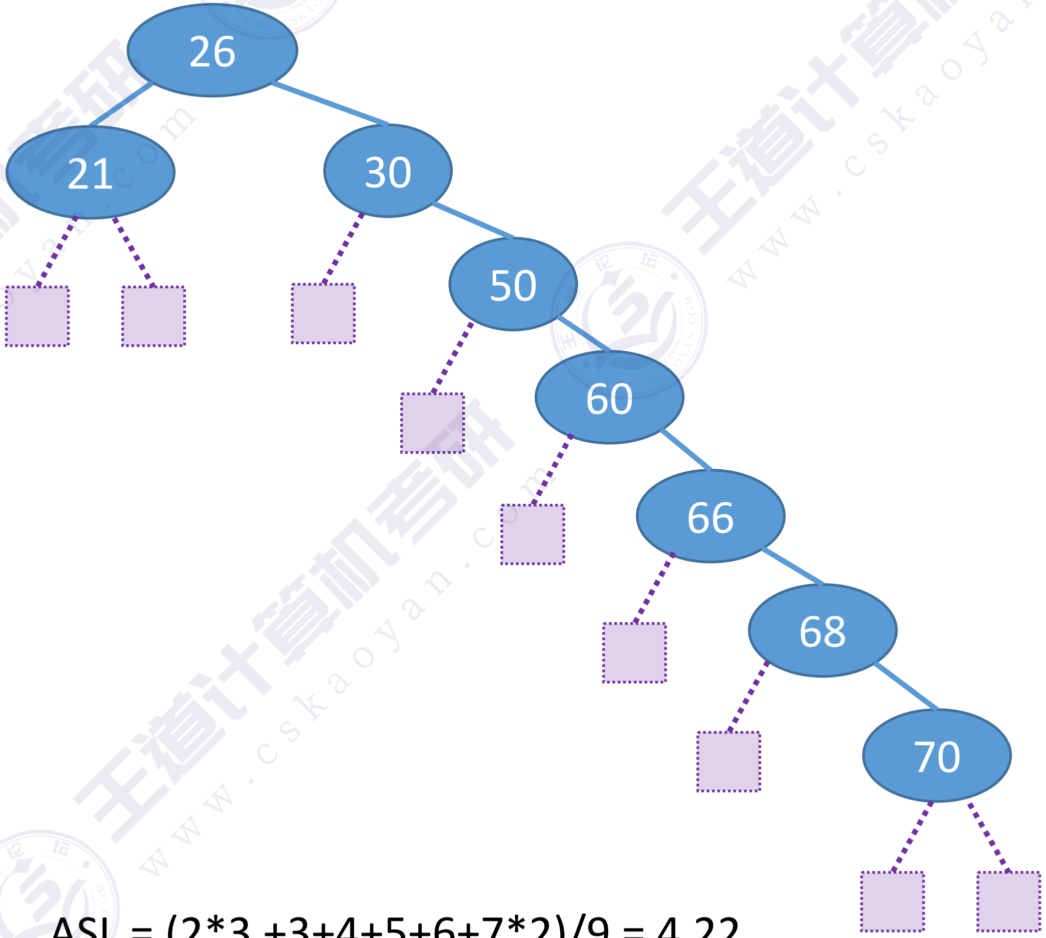

str={26, 21, 30, 50, 60, 66, 68, 70}


n个结点的二叉树最小高度为$\lfloor \log_2 n \rfloor + 1$（完全二叉树）
而平衡二叉树高度与完全二叉树同等数量级

# 查找效率分析

查找长度——在查找运算中，需要对比关键字的次数称为查找长度。




查找失败的平均查找长度 **ASL**（**Average Search Length**）

ASL = (3*7 + 4*2)/9 = 3.22

ASL = (2*3 +3+4+5+6+7*2)/9 = 4.22

# 知识回顾与重要考点

- 二叉排序树
  - 二叉排序树的定义
    - 左子树结点值 < 根结点值 < 右子树结点值
    - 默认不允许两个结点的关键字相同
  - 查找操作
    - 从根节点开始，目标值更小往左找，目标值更大往右找
  - 插入操作
    - 找到应该插入的位置（一定是叶子结点），一定要注意修改其父节点指针
  - 删除操作
    - ①被删结点为叶子，直接删除
    - ②被删结点只有左或只有右子树，用其子树顶替其位置
    - ③被删结点有左、右子树
      - 可用其后继结点顶替，再删除后继结点
      - 或用其前驱结点顶替，再删除前驱结点
      - 前驱：左子树中最右下的结点
      - 后继：右子树中最左下的结点
  - 查找效率分析
    - 取决于树的高度，最好O(log n)，最坏O(n)
    - 平均查找长度的计算
      - 查找成功的情况
      - 查找失败的情况（需补充失败结点）